Theoretical Computer Science 92 (1992) 181–189
Elsevier

Minimisation of acyclic deterministic automata in linear time

Dominique Revuz

CERIL, 25 cours Blaise Pascal, 91000 Evry, France

Abstract.

Revuz, D., Minimisation of acyclic deterministic automata in linear time, Theoretical Computer Science 92 (1992) 181–189.

We present a linear algorithm for the minimisation of acyclic deterministic automata. This algorithm can be used, in particular, on automaton representing lexicons.

1. Introduction

An acyclic automaton is one such that the underlying graph is acyclic. Acyclic automata are used in practice on various problems, and are a very efficient data structure for lexicon representation, see [2, 7, 10]. Access time is linear in the word size and compression results are excellent, minimisation being the best size saving operation. Another field of application is boolean function manipulation; using minimal automata speeds up algorithms such as satisfiability test, equivalence test and function composition, see [3]. For other related work we refer to [4–6, 9].

The minimisation of an n-state deterministic automaton is known to be realisable in time $O(n \log n)$ by an algorithm due to Hopcroft; see [1, 2] for a description. This algorithm is a refinement of the classical $O(n^2)$ algorithm due to Moore [8].

Our algorithm uses the idea presented in [1, example 3.2] to test tree isomorphism using a renumbering scheme to minimise the time and place used by the lexicographic sort.

A short sketch of the algorithm: Each state of the DAWG is labelled with a string describing the reduced automata starting at this state. Working up in increasing levels (level = longest distance to a terminal statc) the algorithm spreads the labels, which are created from the labels of its following states. At each level a lexicographic sort is applied to the list of labels and all states with identical labels are merged (such states

0304-3975/92/$05.00

are equivalent, whence a minimisation). The labelling is done once and only once on each state and the sorting is linear; so, the overall complexity is linear in the number of transitions.

Some notions and notation are listed in the next section; Section 3 describes an acyclic automata minimisation algorithm with any independent sorting scheme, Section 4 is dedicated to sorting schemes and their adaptation to our specific case, and Section 5 gives the final algorithm.

2. Notation

Finite sequences of letters on Σ will be called words. A language is a subset of Σ^*, the set of all words on Σ. If xy is a word then x is a prefix and y a suffix. If X is a sequence of words we call common prefix of X the set of words which are the longest prefix of at least one pair of words from X.

Example. If $X=\{a,b,c,d\}$, then the common prefix of the sequence X is the set reduced to the empty word. If $X=\{ab, aba, abc, bac, cab\}$ the total length of the common suffix is $|ab|+|ab|+|ab|=6$. The total length of the words of this set will be useful to define the complexity of the sorting algorithm.

A DAWG (directed acyclic word graph) or automaton $\mathscr{A}$ is defined by the following 5-uple:

$\mathscr{A}=(Q, \Sigma, F, T, q_0)$, where

- Q is a set of states;
- Σ is an alphabet of finite cardinal denoted by $|\Sigma|$;
- q_0 is the initial state;
- T is the subset of terminal states of Q;
- F is a function of $Q\times\Sigma$ into Q defining the transitions (arcs) of the automaton.

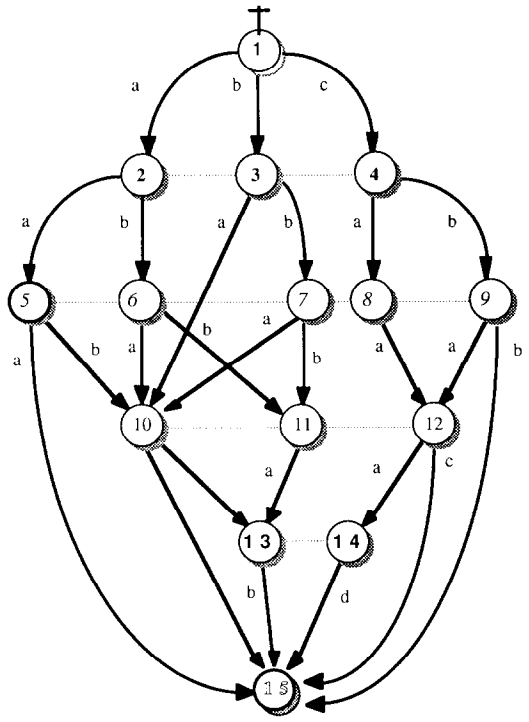
The state reached by the transition of label a of the state q is denoted by $q.a=F(q, a)$. The notation is transitive; if w is a word then $q.w$ denote the state reached by using the transitions labelled by each letter $w_1, w_2, \ldots, w_n$ of w. A word w is accepted by the automaton if $q_0.w$ is in T.

We define the language $L(\mathscr{A})$ recognised by an automaton $\mathscr{A}$ as the set of words w such that $q_0.w$ is final.

An automaton is acyclic if the underlying graph is acyclic. Acyclic automata have associated languages that are finite sets of finite words.

Two automata are said to be equivalent if and only if they recognise the same language. Two states p and q in a given automaton are said to be equivalent if and only if the automaton defined with p and q as initial states are equivalent. Or similarly, if for every word w states $p.w$ and $q.w$ are final then the two states are equivalent.

The dual notion: two states p and q are said to be *distinguished* if and only if there exists a word w such that $p.w$ is final and $q.w$ is not, or $q.w$ is final and $p.w$ is not.



If $\mathscr{A}$ is an automaton there exists a unique automaton $\mathscr{M}$ minimal by the number of states, recognising the same language, i.e. $L(\mathscr{A})=L(\mathscr{M})$. An automaton with no pair of equivalent states is minimal. The minimal automaton for a given language L is the unique automaton with the smallest number of states among those recognising L.

An acyclic automaton $\mathscr{A}$ with $L(\mathscr{A})=\{aa, aaa, aaba, aabbb, abaa, ababb, abbab, baa, babb, abaa, bbbab, caaad, caac, cbaad, cbac, cbb\}$, state 1 is the initial state, states 5, 14, 15 are the final states. States of same height are in the same letter style and connected with a small dotted line. This automaton is not minimal, states *6* and *7* of height 3 are equivalent, to merge states *6* and *7* the edge $(2, b, 6)$ is changed to $(2, b, 7)$. (See Fig. 1.)

3. The minimisation algorithm

We first need to define the *height function* in an acyclic automata: For a state s in the automaton we put $h(s)=\max\{|w| \mid s.w \text{ is final}\}$. The height of the state s is the length of the longest path starting at s and going to a final state. This height function gives a partition Π of Q; Π_i will denote the set of states of height i. We will say that the set Π_i is *distinguished* if no pair of states in Π_i are equivalent.

Now we can state the *height property*.

If every Π_j with $j<i$ is distinguished then two states p and q in Π_i are equivalent if and only if for any letter a in Σ the equality $q.a=p.a$ holds.

Proof. Let p and q be two states in Π_i. We have two possibilities:
- If $p.a$ and $q.a$ are in the same Π_j; since $j<i$ (the automaton being acyclic) by hypothesis the states in Π_j are *distinguished* and, therefore, the states p and q are *distinguished.*
- If $p.a$ is in Π_j and $q.a$ in $\Pi_k, j\neq k$. Suppose without loss of generality that $k<j$; then by the definition of Π there is a word of length j such that $(p.a).w$ is final and $(q.a).w$ is not. So, the states $p.a$ and $q.a$ are distinguished and, hence, so are p and q. □

A simple minimisation algorithm follows from the height property. First we must create the partition by height which is calculated by a standard traversal of the automaton. Time complexity of the traversal function is $O(e)$, where e is the number of edges in our automaton. If the automaton is not a tree, some speedup can be realised with a flag showing that the height of a state was already computed.

Remark. Useless states have no height and should be eliminated in the traversal function.

Minimisation:
calculate Π
For $i:=0$ to $h(q_0)$ **do**
begin
sort the states of Π_i by their edges
merge all equivalent states.
end.

Theorem 1. *Using a sorting scheme with a time complexity* $O(f(n))$, *the preceding algorithm minimises an acyclic automaton in* $O(\sum f(|\Pi_i|))$. *Overall complexity is* $O(e+\sum f(|\Pi_i|))$.

Proof. The correctness of the algorithm comes from the height property. The time bound is obtained by straight computing, thanks to the assumption that merging is integrated in the sorting method. □

4. Lexicographic sort

The implementation of the lexicographic sort we will use is straightforward because we do not need to sort but only to *distinguish* the states two by two. Lexicographic sort is composed of repeated bucket sorts.

Bucket sort

We sort a sequence $a_1, a_2, \ldots, a_n$ of integers $1 \leqslant a_i \leqslant m$:

(1) Create an array of empty queues: $Q[1..m]$

(2) Scan the sequence, $a_1, a_2, \ldots, a_n$ placing the element a_i in $Q[a_i]$, the a_ith bucket.

(3) The concatenated queue of Q make up the sorted list.

The time complexity is $O(n+m)$, the size needed is $O(n+m)$.

Lexicographic sort

We sort a sequence $A_1, A_2, \ldots, A_n$ of k-uplets $(a_{i1}, a_{i2}, \ldots, a_{ik})$, where a_{ij} is in the range 1 to m. We first "bucket sort" the sequence by the kth integer of each k-uplet. The new sequence, a permutation of the first, is ordered according to the kth integer. The following bucket sort on the $(k-1)$th integer keeps this ordering in the buckets. So, by induction, the final sequence, after k bucket sort, is a sorted permutation of the initial sequence.

The time complexity is $O(k(n+m))$, the size needed is $O(n+m)$.

The generalisation to sequences of varying size uplets in the range 1 to $L_{\max}$ is done by bucket sorting the uplet by their length and then executing a lexicographic sort as described, with the additional action of concatenating the uplets of length L to the beginning of the sequence before the $(L_{\max}\text{-}L)$th bucket sort, those strings are sorted by the inexistence of a $(L+1)$th letter.

A complete description of this algorithm is given in [1]. This algorithm can be speeded up with an application of bucket sorts from left to right but it needs an intricate management of nonempty buckets to keep the sequence ordered; we will not go into the details here, but we will use the left-right paradigm in the following *distinguishing* algorithm:

Input: the sequence $A_1, A_2, \ldots, A_n$ of strings $(a_1, a_2, \ldots, a_{k_i}, \$)$, where k_i is positive and a_j is in the range 1 to m.
Output: LEQUAL is a list of equal strings.
(In the minimisation algorithm, equal strings will mean equivalent states.)

```
place A1, A2, ..., An into LIST
place LIST into QUEUE2
i:=0;
 1 repeat
 2 move QUEUE2 to QUEUE1; i:=i+1;
 3 while QUEUE1 not empty do
   begin
 4 Let L be the first list in QUEUE1
 5 while L not empty do
 6   begin
     Let A be the first uplet in L
     if Q[ai] is empty add ai to NONEMPTY
```

```
 7      move A to bucket Q[a_i]
       end
 8     for every bucket in NONEMPTY with more than one element
         add the bucket as a list to QUEUE2
 9     remove buckets with only one string.
10     add Q[$] to LEQUAL
11     end.
until QUEUE2 not empty
```

Theorem 2. *The preceding algorithm distinguishes a sequence of n, uplets of varying lengths where each component of an uplet is an integer between* 1 *and m, in time* $O(n')$, *where n' is the total length of the common prefix of the sequence. An additional memory of size* $O(n+m)$ *is needed.*

Proof. The proof that the algorithm works correctly is by induction on the number of executions of the outer loop. The induction hypothesis is that after r executions of the outer loop, each list in QUEUE2 is made only of uplets with a common prefix of length r.

In the inner loop a list L is split by the ith integer of each uplet, thus creating one or more list in which the ith integer is a constant. So, after the inner loop has been gone through, the lists added to QUEUE2 have the induction property. This inner loop is executed on every list in QUEUE1; so, the induction property holds.

The program terminates because the length of the longest common suffix is less than or equal to the maximal length of the words of the sequence.

Step 8 takes at most the same number of steps as the inner loop (in the worst case a list is created with every element of L); so, our time complexity is bounded by the number of times step 7 is executed and this is exactly n', namely, the total length of the words in the common prefix of the sequence. From this follows the $O(n')$ time bound. □

The additional memory is for list pointers used in QUEUEs and the bucket array.

5. The final algorithm

We now try to merge the two preceding algorithms to obtain a linear algorithm: we first must see the states as uplets or strings. Thus, we label each state s with the following:

$$label(s) = (\text{F or NF}, l_1, nl_1, l_2, nl_2, \ldots, l_k, nl_k),$$

where F or NF tells whether the state s is final and where for each of the edges of state s in lexicographic order, l_i is the letter on the ith edge and nl_i the name of the state pointed by the ith edge.

With this labelling scheme we obtain a time complexity of $O(\sum|l'_i|+e)$, where l'_i is the total length of the common prefix of the labels of height i. We must bound those l'_i which depends on the range of the nl_i whose values are state numbers, the length of which is bounded by $\log(|Q|)$ (representation with digits).

In the lexicographic sort, the state numbers can be seen as letters or as strings:
- in the first case the buckets will have to be of size $|Q|$
- in the second case the length of the label is enlarged by a $\log(|Q|)$ factor; the time bound will not be linear.

We lower the two bounds with the following renumbering:

To renumber the states we label them with a pair (*current_height*, *number*). When a state number is needed by the sorting algorithm, i.e. a split is done on a nl_i. A new name is used in the place of nl_i computed in the following way.

Suppose we are sorting *current_height* level states, then two cases arise when a state number is needed in the bucket sort:

In the pair (*ch*, *num*) in state nl_i the *ch* value is different from the *current_height*, then the pair (*ch*, *num*) is an old pair we must put in its place the new pair (*current_height*, *new_num*).

In the case where *ch* equal *current_height* the part *num* of the pair is directly used.

```
function renumber(s, h, n)
begin
if (s→ch! = h) {s→ch:= h; s→num:= n; n:= n + 1;}
return (s→num);
end;
```

Now our bucket array is bounded by the maximum of $|\Sigma|$ and E_i, where E_i is the total number of edges of Π_i-states.

This renumbering technique permits to apply any lexicographic sorting algorithm and obtain a linear complexity of the label length of the order of $|E_i|$ and not $|E_i|\log|Q|$. In our case the renumeration technique only reduces the bucket array size.

Final algorithm:

Compute h for every state with the traversal function and create the sets Π_i
Merge all Π_0 states (they are all equivalent).

```
For i:=1 to h(q0)−1 do
begin
  put Πi as a list into QUEUE2          (1)
i:=0;
do
move QUEUE2 to QUEUE1; i:=i+1;
while QUEUE1 not empty do
  begin
  Let L be the first list in QUEUE1
```

```
  while L not empty do
    begin
    S be the first state in L
    move S to bucket number (Q[a_i])
  end
  for every bucket with more than one element
  add the bucket as a list to QUEUE2
  remove buckets with only one string.
  merge all states of Q[$]
  end.
while QUEUE2 not empty
end.
```

6. Some implementation remark

To obtain a faster algorithm the labels can be extended to

$$label(s) = (\text{F or NF}, nbe, l_1, nl_1, l_2, nl_2, \ldots, l_k, nl_k),$$

where *nbe* is the number of edges and the traversal function can split Π_i in $F\Pi_i$ and $N\Pi_i$, where F and N means final and nonfinal states. The program is changed at line (1) where it becomes

put $F\Pi_i$ and $N\Pi_i$ as two lists in QUEUE2.

7. Conclusion

In practice, for small automata the sorting scheme to use is the easiest to program; when working on huge automata (we work on a 550 000 words dictionary and are looking forward to work on 1 Million states automaton) where the minimisation algorithm must be repeatedly used to compress the automaton (so as to keep it in memory), the distinguishing algorithm is a must.

The representation of finite sets by acyclic deterministic automaton is a good general purpose data structure, the member function is linear in the word length and independent of the dictionary size, the insert and delete functions are linear in length of the inserted or deleted word (but the automaton is not always minimal after such functions and a minimisation can be called as needed).

The most interesting aspect of automaton representation is the size-saving property. For example a 300 000 words dictionary taking 5.2 M bytes in text format was compacted to 0.3 M bytes in automaton format, a 0.06 compression ratio.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).

[2] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, Principles, Techniques and Tools* (Addison-Wesley, Reading, MA, 1986).

[3] R.E. Bryant, Graph-based algorithms for boolean function manipulation, *IEEE Trans. on Computers*, **C-35**, (5) 677–691.

[4] A. Cardon and M. Crochemore, Partitioning a graph in $O(|A|\log_2|V|)$, *Theoret. Comput. Sci.* **19** (1982) 85–98.

[5] J.E. Hopcroft and R.M. Karp, An algorithm for testing equivalence of finite automata, TR-71–114, Dept. of Computer Science, Cornell Univ. 1971; see [1] 143–145 for a description.

[6] D.A. Huffman, The synthesis of sequential switching machines, *J. Franklin Inst.* **257** (1954) 275–303.

[7] M. Minsky, *Compilation, Finite and Infinite Machines* (Prentice Hall, Englewood Cliffs, NJ, 1967).

[8] E.F. Moore, *Gedanken Experiments on Sequential Machines: Automata studies* (Princeton University Press, Princeton, NJ, 1956) 129–153.

[9] R. Paige and R.E. Tarjan, Three partition refinement algorithms, *SIAM J. Comput.* **16** (6) (1987) 973–989.

[10] M.O. Rabin and D. Scott, Finite automata and their decision problems, *IBM J. Res. Develop.* **3** (1959) 114–125.